Instruction Level Parallelism

Introduction

All processors since about 1985 use *pipelining* to overlap the execution of instructions and improve performance. This potential overlap among instructions is called **Instruction-Level Parallelism** (ILP), since the instructions can be evaluated in parallel.

There are two largely separable approaches to exploiting ILP:

1. An approach that relies on hardware to help discover and exploit the parallelism dynamically
2. An approach that relies on software technology to find parallelism statically at compile time

Processors using the first approach dominate in the desktop and server markets, while in the Personal Mobile Device (PMD) market, where energy efficiency is often the key objective, designers exploit the second approach more.

The amount of parallelism within a *basic block* (a straight-line code sequence with no branches in except to the entry and no branches out except at the exit) is quite small. For a typical MIPS program, the average dynamic branch frequency is often between 15% and 25%, meaning that between three and six instructions execute between a pair of branches. Since these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be less than the average block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

Loop-level parallelism

The simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop. This type of parallelism is often called *loop-level parallelism.*

Here is a simple example of a loop that adds two 1000-element arrays and is completely parallel:

```
for (int i = 0; i <= 999; i++) {
        x[i] = x[i] + y[i];
}
```

There are different techniques to convert such loop-level parallelism into instruction-level parallelism. Basically, they entail the unrolling of the loop either statically (as in done by the compiler) or dynamically (as in done by the hardware).

An important, alternative method for exploiting loop-level parallelism is the use of SIMD in both vector processors and Graphics Processing Units (GPUs). A SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel, while a vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline.

Loop unrolling: exposing ILP

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline.

To avoid a pipeline stall, the execution of a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline.

In this subsection, we look at how the compiler can increase the amount of available ILP by transforming loops. We rely on the following code segment, which adds a scalar to a vector:

```
for (int i = 0; i < 999; i++) {
        x[i] = x[i] + s;
}
```

We can see that this loop is parallel by noticing that the body of each iteration is independent.

If we translate this code in MIPS and we avoid scheduling it for our pipeline, we obtain:

```
Loop:   L.D    F0, 0(R1)    ;F0=array element
            ADD.D  F4, F0, F2   ;add scalar in F2
            S.D    F4, 0(R1)    ;store result
            DADDUI R1, R1, #-8  ;decrement pointer by 8 bytes
            BNE    R2, R2, Loop ;branch R1 ≠ R2
```

When scheduled, this loop will run in nine clock cycles, adding four stalls in its execution. If we schedule the loop to optimise it, we can bring this figure down to seven clock cycles, adding just two stalls.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is **loop unrolling**. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instruction from different iterations to be scheduled together. In this case, we can eliminate the data use stalls by creating additional independent instructions within the loop body.

In real programs, we do not usually know the upper bound on the loop. Suppose it is n, and we would like to unroll the loop to make k copies of the body. Instead of a single unrolled loop, we generate a paid of consecutive loops. The first executes $(n \bmod k)$times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates (n/k) times. For large values of n, most of the execution time will be spent in the unrolled loop body.

In the previous example, unrolling improves the performance of this loop by eliminating overhead instructions, although it increases code size substantially. Still, the gain from scheduling on the unrolled loop is even larger than on the original loop.

Three different effects limit the gains from loop unrolling:

1. A decrease in the amount of overhead amortised with each unroll
2. Code size limitations
3. Compiler limitations

Let's consider the question of loop overhead first. When we unrolled the loop four times, it generated sufficient parallelism among the instructions that the loop could be scheduled with no stall cycles. If the loop is unrolled eight times, the overhead is reduced from $1/2$ cycle per original iteration to $1/4$.

A second limit to unrolling is the growth in code size that results. For larger loops, the code size growth may be a concern particularly if it causes an increase in the instruction cache miss rate.

Another factor, often more important than code size, is the potential shortfall in registers that is created by aggressive unrolling and scheduling. This secondary effect that results form instruction scheduling in large code segments is called **register pressure**. It arises because scheduling code to increase ILP causes the number of live values to increase. After aggressive instruction scheduling, it may not be possible to allocate all the live values to registers. The transformed code, while theoretically faster, may lose some or all of its advantage because it generates a shortage of registers.